

# WATOR: un simulatore distribuito di un modello biologico

Progetto del modulo di laboratorio SOL 2014/15

## Contents

<b>1</b>	<b>Introduzione</b>	<b>1</b>
1.1	Materiale in linea . . . . .	2
1.2	Struttura del progetto e tempi di consegna . . . . .	2
1.3	Valutazione del progetto di recupero . . . . .	2
<b>2</b>	<b>Il progetto: WATOR</b>	<b>3</b>
<b>3</b>	<b>Il processo wator</b>	<b>4</b>
<b>4</b>	<b>Il processo visualizer</b>	<b>6</b>
4.1	Messaggi da wator a visualizer . . . . .	7
<b>5</b>	<b>Lo script watorsript</b>	<b>7</b>
<b>6</b>	<b>Istruzioni</b>	<b>7</b>
6.1	Materiale fornito dai docenti . . . . .	7
6.2	Cosa devono fare gli studenti . . . . .	8
<b>7</b>	<b>Parti Opzionali</b>	<b>8</b>
<b>8</b>	<b>Codice e documentazione</b>	<b>8</b>
8.1	Vincoli sul codice . . . . .	8
8.2	Formato del codice . . . . .	8
8.3	Relazione . . . . .	9

## 1 Introduzione

Il modulo di Laboratorio di Programmazione di Sistema del corso di Sistemi Operativi e Laboratorio (277AA) prevede lo svolgimento di un progetto individuale suddiviso in tre *frammenti*. Questo documento descrive la struttura complessiva del progetto e dei vari frammenti che lo compongono. Il progetto viene sviluppato e documentato utilizzando gli strumenti, le tecniche e le convenzioni presentati durante il corso.

## 1.1 Materiale in linea

Tutto il materiale relativo al corso può essere reperito sul sito Web:

<http://www.cli.di.unipi.it/doku/doku.php/informatica/sol/>

Il sito verrà progressivamente aggiornato con le informazioni riguardanti il progetto (es. FAQ, suggerimenti, avvisi), il ricevimento e simili. Il sito è un Wiki e gli studenti possono registrarsi per ricevere automaticamente gli aggiornamenti delle pagine che interessano maggiormente. In particolare consigliamo a tutti di registrarsi alla pagina delle 'FAQ' e degli 'Avvisi Urgenti'.

Eventuali chiarimenti possono essere richiesti consultando i docenti di del corso durante l'orario di ricevimento, le ore in laboratorio e/o per posta elettronica.

## 1.2 Struttura del progetto e tempi di consegna

Il progetto deve essere sviluppato dallo studente individualmente e può essere consegnato entro il 15 Gennaio 2016.

La consegna del progetto avviene *esclusivamente* inviando per posta elettronica il tar creato dal target **consegna3** del **Makefile** contenuto nel kit di sviluppo del progetto. Il tar deve essere allegato ad un messaggio con soggetto

**lso15: Consegna Progetto Finale**

ed inviato al docente del corso all'indirizzo di posta elettronica fornito sulla pagina web. Le consegne sono seguite da un messaggio di conferma da parte del docente all'indirizzo di mail da cui la consegna è stata effettuata. Se la ricezione non viene confermata entro 3/4 giorni lavorativi, contattare il docente per e-mail.

*I progetti che non contengono tutti i file necessari o che non compilano ed eseguono correttamente su versioni recenti di Ubuntu non saranno accettati. I progetti che non rispettano il formato o non generati con il target **consegna3** non verranno accettati. Si prega di controllare che tutti i file necessari alla corretta compilazione ed esecuzione del progetto siano presenti nel tar prima di inviarlo.*

La data ultima di consegna è il 15/01/2016. Dopo questa data gli studenti dovranno svolgere il nuovo progetto previsto per il corso 2015/16.

Inoltre, il progetto è suddiviso in tre frammenti. Gli studenti che consegnano una versione sufficiente di ogni frammento entro la data di scadenza specificata sul WEB accumulano dei bonus di 2 punti che contribuiscono al voto finale con le modalità illustrate nelle slide introduttive del corso (vedi sito).

## 1.3 Valutazione del progetto di recupero

Al progetto viene assegnata una valutazione da 0 a 30 di cui 6 punti esclusivamente determinati dalla *qualità della documentazione allegata*. La valutazione del progetto è effettuata in base ai seguenti criteri:

- motivazioni, originalità ed economicità delle scelte progettuali
- strutturazione del codice (suddivisione in moduli, uso di makefile e librerie etc.)
- efficienza e robustezza del software
- modalità di testing
- aderenza alle specifiche
- qualità del codice C e dei commenti
- chiarezza ed adeguatezza della relazione (vedi Sez. 8.3)

La prova orale tenderà a stabilire se lo studente è realmente l'autore del progetto consegnato e verterà su tutto il programma del corso e su tutto quanto usato nel progetto anche se non fa parte del programma del corso. L'esito della prova orale è essenziale per delineare il voto finale. In particolare, l'orale comprenderà

- una discussione delle scelte implementative del progetto
- l'impostazione e la scrittura di semplici script bash e makefile
- l'impostazione e la scrittura di programmi C + PosiX non banali (sia sequenziali che concorrenti)
- domande su tutto il programma presentato durante il corso.

**Casi particolari** Gli studenti lavoratori iscritti alla laurea triennale in Informatica possono consegnare i due frammenti e il progetto finale in un'unica soluzione ottenendo tutti i bonus in qualsiasi appello dell'anno. In questo caso è necessaria la certificazione da consegnare al docente come da regolamento di ateneo.

Gli studenti che svolgono il progetto per le lauree di secondo livello sono invitati a contattare il docente.

Gli studenti iscritti ai vecchi ordinamenti (nei quali il voto di Lab. 4 contribuiva al voto di SO) avranno assegnato un voto in trentesimi che verrà combinato con il voto del corso di SO corrispondente secondo modalità da richiedere ai due docenti coinvolti.

## 2 Il progetto: WATOR

Lo scopo del progetto è lo sviluppo di un sistema distribuito che simula un pianeta acquatico in base ad un semplice modello del comportamento usato in etologia <sup>1</sup>. Il modello utilizza un ipotetico pianeta (*Wator*) ricoperto interamente da un oceano temperato. L'oceano è abitato da predatori (gli *squali*) e prede (i *pesci*). Wator viene rappresentato da una matrice rettangolare. Ogni elemento della matrice rappresenta un'area dell'oceano che può essere in tre stati distinti

- può essere vuota [WATER]
- può contenere esattamente un pesce [FISH]
- può contenere esattamente uno squalo [SHARK]

Siccome Wator è un pianeta rotondo le celle della prima riga (colonna) sono adiacenti alle celle dell'ultima riga (colonna).

Il tempo è a sua volta suddiviso in unità standard dette *chronon*, il sistema si evolve in passi che corrispondono ad un chronon ciascuno. Ad ogni passo la matrice che rappresenta il pianeta viene aggiornata secondo le seguenti regole:

- *gli squali mangiano e si spostano* A ogni passo, uno squalo presente in una cella  $(i, j)$  si guarda intorno nelle celle adiacenti. Se una di queste celle contiene un pesce, lo squalo mangia il pesce (la scelta è casuale) e si sposta nella cella precedentemente occupata dal pesce. Se nessuna delle celle adiacenti contiene un pesce, lo squalo si sposta in una delle celle adiacenti vuote (la scelta è casuale). Le celle adiacenti sono le celle in cui uno degli indici differisce esattamente di uno da  $i$  e  $j$ . Cioè  $(i-1, j)$ ,  $(i, j-1)$ ,  $(i, j+1)$ ,  $(i+1, j)$ . Graficamente

$$\begin{array}{ccccc} & & (i-1, j) & & \\ & (i, j-1) & *** & (i, j+1) & \\ & & (i+1, j) & & \end{array}$$

<sup>1</sup>A. K. Dewdney, Sharks and fish wage an ecological war on the toroidal planet Wator, Scientific American, December 1984

- *gli squali si riproducono e muoiono* Se uno squalo sopravvive per almeno  $s_b$  chronon si riproduce: se é disponibile una cella adiacente vuota questa viene occupata da un nuovo squalo. Lo squalo che si é appena riprodotto potrà farlo di nuovo dopo altri  $s_b$  chronon. Se uno squalo non mangia per  $s_d$  chronon consecutivi muore e scompare dalla griglia.
- *i pesci si spostano* A ogni passo un pesce si sposta casualmente in una delle celle adiacenti. Siccome Wator é un pianeta rotondo i pesci (e gli squali) possono nuotare verso l'esterno di un lato della matrice e risbucare dal lato opposto.
- *i pesci si riproducono* Se un pesce sopravvive per almeno  $f_b$  chronon si riproduce: se é disponibile una cella adiacente vuota questa viene occupata da un nuovo pesce. Se tutte le celle adiacenti sono occupate non si riproduce. Il pesce che si é appena riprodotto potrà farlo di nuovo dopo altri  $f_b$  chronon.

All' inizio della simulazione, la griglia deve essere riempita casualmente con  $n_s$  squali e  $n_f$  pesci. Ogni squalo (pesce) é munito di un contatore, inizializzato a zero ed incrementato ad ogni chronon. Quando il valore del contatore raggiunge  $s_b$  ( $f_b$ ) lo squalo (il pesce) si riproduce ed il contatore viene di nuovo azzerato.

I pesci e gli squali applicano sempre prima la regola relativa alla riproduzione e poi quella relativa a mangiare e spostarsi. In particolare gli squali prima cercano sempre di mangiare un pesce. I pesci e gli squali i appena nati vengono aggiornati solo al chronon successivo.

Questo modello, per quanto semplice, é sufficiente a fornire un comportamento simile a quello che viene osservato davvero in alcune popolazioni in natura. Se c' é abbastanza pesce, gli squali prosperano e si riproducono finché non raggiungono un numero troppo elevato. A questo punto quasi tutto il pesce verrà divorato e inizierà un periodo di carestia.

Il sistema è costituito da due processi diversi:

- **wator**: il processo che si occupa di eseguire la simulazione distribuita
- **visualizer**: il processo che si occupa di visualizzare a video il pianeta e la sua popolazione ogni C chronon.

**wator** e **visualizer** sono i due processi che devono essere realizzati nel progetto didattico. I processi comunicano via socket AF\_UNIX. Per rendere più semplice lo sviluppo ed il testing dell'applicazione sulle macchine del centro di calcolo per la didattica tutte le socket vengono create nella directory locale `./tmp` invece che in `/tmp` come sarebbe logico aspettarsi. Quest'ultima soluzione renderebbe infatti possibili interazioni indesiderate e fastidiose fra progetti sviluppati da utenti diversi sulla stessa macchina.

Inoltre, deve essere realizzato uno script bash (**watorscript**) come specificato in Sez. 5.

### 3 Il processo wator

Il processo wator realizza internamente una tipica struttura a *farm* utilizzando tre tipi di thread

- thread **dispatcher**: che si occupa di generare i task relativi alla simulazione distribuita, ciascuno relativo alla simulazione di una parte del pianeta
- thread **worker**: che si occupa di eseguire task dalla sequenza generata dal dispatcher
- thread **collector**: che si occupa di controllare che la simulazione di tutti i task relativi alla matrice di un chronon siano stati elaborati e di inviare eventualmente informazioni al processo visualizzatore per mostrare lo stato attuale a schermo.

**wator** viene attivato da shell con il comando

```
sd x1
sb x2
fb x3
```

Figure 1: Formato del file `wator.conf`

```
nrow
ncol
w[0][0] ..... w[0] [ncol-1]
w[1][0] ..... w[1] [ncol-1]
.....
w[nrow-1][0] ..... w[nrow-1] [ncol-1]
```

Figure 2: Formato del file che descrive lo stato iniziale del pianeta.

```
$ wator file [-n nwork] [-v chronon] [-f dumpfile]
```

tutti i parametri relativi alla simulazione sono specificati da un file di configurazione `wator.conf`, che è un file di testo con il formato mostrato in Fig. 1, dove ogni riga ha come primi due caratteri una fra le coppie `sd,sb,fb` ed un valore intero positivo. I primi due caratteri individuano i corrispondenti parametri del modello ed il valore successivo specifica il valore del parametro da usare nella simulazione. Ogni riga è terminata da un newline (`\n`). I valori di  $n_f$  ed  $n_s$  della simulazione vengono invece settati implicitamente dalla configurazione iniziale del pianeta in `file`, descritta più avanti. Se il file `wator.conf` non esiste il processo termina stampando un messaggio di errore. Le opzioni del comando `wator` possono comparire in un ordine qualsiasi sulla linea di comando. La opzione `-n` specifica il numero di worker da usare e l'opzione `-v` specifica ogni quanti `chronon` la configurazione del mondo deve essere visualizzata (i valori di default delle due opzioni sono scelti dallo studente e settati usando due macro `NWORK_DEF` e `CHRON_DEF`). L'opzione `-f` specifica che il processo `visualizer` dovrà effettuare il dump del pianeta sul file `dumpfile` invece che fare la visualizzazione a video (il file `dumpfile` viene sempre sovrascritto dal `visualizer` e dovrà avere lo stesso formato del file mostrato in Fig. 2). La configurazione iniziale del mondo viene caricata dal file `file` il cui formato è mostrato in Fig. 2. Si tratta di un file testuale in cui nelle prime due righe sono specificati due interi positivi che rappresentano il numero di righe (`nrow`) ed il numero delle colonne (`ncol`) della matrice. Nelle linee successive sono indicati ordinatamente tutti i valori della matrice una riga per linea. I valori sono separati da un blank e ogni riga è terminata da un newline. Fig. 2 mostra un file che rappresenta un pianeta con 3 righe e 7 colonne in cui  $n_f = 6$  e  $n_s = 4$ , i caratteri `W`, `F` ed `S` rappresentano acqua, pesci e squali rispettivamente.

L'opzione `-n` permette di specificare il numero `nwork` di thread worker da utilizzare nell'implementazione. L'opzione `-v` specifica ogni quanti `chronon` lo stato del pianeta deve essere visualizzato. La visualizzazione avviene aprendo una connessione con la socket `visual.sck` che connette al processo `visualizer` con il protocollo specificato in Sez. 4.1.

Le opzioni possono apparire in un ordine qualsiasi nella riga di comando.

Una volta attivato `wator` carica da `file` la matrice che rappresenta il pianeta, attiva il processo `visualizer` e attiva un thread dispatcher, un thread collector ed `n` thread worker. Ogni worker è identificato da un numero da 0 a `nwork-1` detto `wid`. Ogni worker elabora un rettangolo `KxN` della matrice dove `K` ed `N` sono specificati da opportune macro. Ogni thread worker appena inizia l'elaborazione crea immediatamente un file vuoto chiamato `wator_worker_wid` in cui `wid` è sostituito dal `wid`

```

3
5
W W F W W
F F S S F
W S S F F

```

Figure 3: Esempio di file che rappresenta un pianeta con 3 righe e 5 colonne con  $n_f = 6$  e  $n_s = 4$ , i caratteri **W**, **F** ed **S** rappresentano acqua, pesci e squali rispettivamente.

del worker. Durante la computazione si applicano le regole esattamente nell'ordine discusso in Sec. 2. L'aggiornamento viene fatto direttamente sulla matrice che rappresenta il pianeta, avendo cura di assicurare che uno squalo (pesce) venga aggiornato esattamente una volta per ogni chronon. Nella relazione, lo studente dovrà specificare le strutture dati e la strategia usata per assicurare l'aggiornamento singolo per chronon di ogni pesce e squalo. Il comportamento dei thread è ciclico. Ad ogni chronon **X** il dispatcher inserisce in una coda FIFO condivisa tutti i rettangoli da elaborare ed i worker estraggono dalla coda i rettangoli da elaborare, calcolano l'aggiornamento per tutte le celle del rettangolo e aggiornano il loro valore nella matrice prima di prelevare dalla coda un nuovo task. L'accesso alla coda ed alla matrice deve essere programmato in modo corretto con opportuni meccanismi di sincronizzazione. La scelta di tali meccanismi è responsabilità dello studente e deve essere descritta e motivata nella relazione.

Il thread collector si occupa di implementare una politica di sincronizzazione in modo da stabilire quando tutti i task che corrispondono ai rettangoli relativi ad un certo chronon sono stati elaborati ed è possibile passare al chronon successivo. In questo caso, avverte il dispatcher di iniziare la nuova elaborazione. Se **X** modulo **chronon** è uguale a 0, il collector si occupa anche di fornire la nuova matrice al processo **visualizer** secondo il protocollo in sez. 4.1.

Il processo alla ricezione di un **SIGUSR1** salva lo stato corrente della matrice su file. Il file si chiama **wator.check** ed il formato utilizzato è quello di Fig. 2. Se il file esiste viene sovrascritto.

Infine, il processo **wator** termina quando riceve un segnale di **SIGINT** (se l'esecuzione è in foreground) o **SIGTERM**, in questo caso si implementa un algoritmo di terminazione gentile in cui i thread worker terminano la computazione relativa al chronon corrente e il visualizzatore visualizza comunque l'ultima iterazione ed entrambi i processi terminano rimuovendo la socket di comunicazione ed eventuali file temporanei. Il file **wator.check** non viene rimosso.

## 4 Il processo visualizer

Il **visualizer** è un processo Unix che viene attivato dal processo **wator** per gestire la visualizzazione del pianeta acquatico. Il **visualizer** appena attivato si mette in attesa di connessioni sulla socket **visual.sck**. Ad ogni connessione seguono uno o più messaggi secondo il protocollo in sec 4.1 che trasmettono i valori del mondo da visualizzare. Dopo la ricezione completa di un pianeta da visualizzare il processo chiude la connessione e si mette in attesa di una nuova connessione.

In fase di terminazione gentile del processo **wator**, anche il processo **visualizer** deve terminare.

## 4.1 Messaggi da wator a visualizer

Il protocollo fra wator e visualizer deve essere fissato dallo studente e documentato nella relazione in modo da

- contenere tutte le informazioni relative alla configurazione del mondo da visualizzare e
- minimizzare il numero di byte spediti

la struttura scelta deve essere motivata e descritta accuratamente nella relazione.

## 5 Lo script watorscript

Lo script esamina un file nel formato mostrato in Fig. 2 estraendo alcune informazioni sulla popolazione del pianeta. Lo script si attiva da linea di comando con diverse opzioni

```
watorscript file
watorscript -s file
watorscript -f file
watorscript --help
```

Senza nessuna opzione lo script controlla soltanto che il formato del file sia corretto e stampa l'esito del controllo (OK/NO) su standard error. Le opzioni possono apparire in un qualsiasi ordine sulla linea di comando, ad esempio le due seguenti invocazioni sono corrette

```
watorscript file -s
watorscript file -f
```

```
bash$ watorscript file
OK
```

Le opzioni `s` ed `f` contano il numero di squali e di pesci rispettivamente e stampano il valore sullo standard output come in

```
bash$ watorscript -s file
18
```

Infine l'opzione lunga `help` stampa un messaggio di uso dello script.

È richiesto di segnalare le situazioni erronee con opportuni messaggi su standard error.

È fatto esplicito divieto di usare `sed` o `awk` nella realizzazione dello script.

## 6 Istruzioni

### 6.1 Materiale fornito dai docenti

Nei kit del progetto vengono forniti

- funzioni di test e verifica
- `Makefile` per test e consegna
- file di intestazione (`.h`) con definizione dei prototipi e delle strutture dati
- vari `README` di istruzioni

## 6.2 Cosa devono fare gli studenti

Gli studenti devono:

- leggere *attentamente* le specifiche del progetto (questo documento) ed i README dei vari frammenti e capire il funzionamento del codice fornito dai docenti
- implementare le funzioni richieste e testarle
- verificare le funzioni con i test forniti dai docenti (attenzione: questi test vanno eseguiti su codice già corretto e funzionante altrimenti possono dare risultati fuorvianti o di difficile interpretazione)
- preparare la *documentazione*: vedi la Sez. 8.
- sottomettere i tre frammenti del progetto esclusivamente utilizzando il makefile fornito e seguendo le istruzioni nel README.

## 7 Parti Opzionali

Possono essere realizzate funzionalità ed opzioni in più rispetto a quelle richieste (ad esempio altre opzioni di wator, visualizzazioni diverse o altre statistiche nello script).

Le parti opzionali devono essere spiegate nella relazione e corredate di test appropriati ove opportuno.

## 8 Codice e documentazione

In questa sezione, vengono riportati alcuni requisiti del codice sviluppato e della relativa documentazione.

### 8.1 Vincoli sul codice

*Makefile e codice devono compilare ed eseguire CORRETTAMENTE su versioni recenti di UBUNTU o su un insieme non vuoto delle macchine del Centro di Calcolo della Didattica. Il README (o la relazione) deve specificare su quali versioni di UBUNTU o su quali macchine CDC è possibile far girare correttamente il codice.*

La stesura del codice deve osservare i seguenti vincoli:

- la compilazione del codice deve avvenire definendo delle regole appropriate nel Makefile;
- il codice deve compilare senza errori o warning utilizzando le opzioni `-Wall -pedantic`
- NON devono essere utilizzate funzioni di temporizzazione quali le `sleep()` o le `alarm()` per risolvere problemi di race condition o deadlock fra i processi/thread. Le soluzioni implementate devono necessariamente funzionare qualsiasi sia lo scheduling dei processi/thread coinvolti
- DEVONO essere usati dei nomi significativi per le costanti nel codice (con opportune `#define` o `enum`)

### 8.2 Formato del codice

Il codice sorgente deve adottare una convenzione di indentazione e commenti chiara e coerente. In particolare deve contenere

- una intestazione per ogni file che contiene: il nome ed il cognome dell'autore, la matricola, il nome del programma; dichiarazione che il programma è, in ogni sua parte, opera originale dell'autore; firma dell'autore.

- un commento all'inizio di ogni funzione che specifichi l'uso della funzione (in modo sintetico), l'algoritmo utilizzato (se significativo), il significato delle variabili passate come parametri, eventuali variabili globali utilizzate, effetti collaterali sui parametri passati per puntatore etc.
- un breve commento che spieghi il significato delle strutture dati e delle variabili globali (se esistono);
- un breve commento per i punti critici o che potrebbero risultare poco chiari alla lettura
- un breve commento all'atto della dichiarazione delle variabili locali, che spieghi l'uso che si intende farne (se non ovvio)

### 8.3 Relazione

La documentazione del progetto consiste nei commenti al codice e in una breve relazione (massimo 15 pagine) il cui scopo è quello di descrivere la struttura complessiva del lavoro svolto. La relazione *deve rendere comprensibile il lavoro svolto ad un estraneo, senza bisogno di leggere il codice se non per chiarire dettagli implementativi. In particolare NON devono essere ripetute le specifiche contenute in questo documento.* In pratica la relazione deve contenere:

- le principali scelte di progetto (strutture dati principali, algoritmi fondamentali e loro motivazioni)
- la strutturazione del codice (logica della divisione su più file, librerie etc.)
- la struttura dei programmi sviluppati
- la struttura dei programmi di test (se ce ne sono)
- l'interazione fra processi ed i relativi protocolli
- le difficoltà incontrate e le soluzioni adottate
- quanto altro si ritiene essenziale alla comprensione del lavoro svolto
- README di istruzioni su come compilare/eseguire ed utilizzare il codice

La relazione deve essere in formato PDF.